CS422 - Programming Language Design

Continuation-Passing Style (CPS) Transformation

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

On Data Versus Control Context

Our previous language definitions were based on a corresponding notion of *state*, which contained the entire *data context* needed in order to evaluate the "current" expression. When defining the meaning of an expression, we traversed it inductively, evaluated each subexpression in appropriate data contexts, collected the side effects, and then calculated the value of the current expression.

Many implementations of programming languages, especially the naive ones, follow a similar methodology. They internally maintain a *control context*, to know how to continue the execution after evaluating a complex (typically a function call) subexpression.

Let us recall the following expression calculating factorial:

```
let rec f n = if n eq 0
              then 1
              else n * f(n - 1)
in f 5
```

Following the control flow of the evaluation of this expression ignoring the details, we can note the following pattern:

```
f(5) => 5 * f(4)
     => 5 * (4 * f(3))
     => 5 * (4 * (3 * f(2)))
     => 5 * (4 * (3 * (2 * f(1))))
     => 5 * (4 * (3 * (2 * (1 * f(0)))))
     => 5 * (4 * (3 * (2 * (1 * 1))))
     => 5 * (4 * (3 * (2 * 1)))
     => 5 * (4 * (3 * 2))
     => 5 * (4 * 6)
     => 5 * 24
     => 120
```

Compare this execution with the execution of the following "iterative" version of the factorial function:

```
let rec f n m = if n eq 0
                then m
                else f (n - 1) (n * m)
in f 5
```

If we follow the control flow of the evaluation of the expression above, ignoring again the details, we can see the following pattern:

```
f(5,1) => f(4,5) => f(3,20) => f(2,60) => f(1,120) => f(0,120) => 120
```

As expected, the second version of factorial can be evaluated much more quickly than its first version by most implementations. That's because, in the first case, the language implementation *needs to store the control context for each function call* and then to re-traverse it. In the second case, each function call contains all the information that it needs to continue its computation; nothing needs to stored to be processed when the function returns!

Control Behavior

The second definition of factorial is faster since *no control context has to be recorded*. This happens because, at any moment, the return value of the function equals the value returned by its recursive call. One can distinguish two types of control behaviors:

- *Recursive control behavior*. When additional control information must be recorded with each function call, and this information must be retained until the call returns.
- *Iterative control behavior*. When only a bounded amount of memory is needed for control information.

Programs manifesting recursive control behavior tend not only to be slower, but also to run out of memory much more quickly than their iterative control behavior variants. Note, however, that it may very well be possible that programs run out of memory even

under iterative control behavior, because functions may need additional space to store their arguments or local names. But, in that case, we say that they ran out of memory because their *data context* became too large; their control context was bound and its maximum size was known at compile time.

The first version of the factorial requires an implementation to record control information at each call because the recursive function is called *in an operand position*, so after the evaluation of the recursive call, one still has to finish the local computation. We will next learn how to eliminate the need for handling control contexts. The underlying guiding principle will be the following:

> Calling of functions does not create control context in the language implementation/definition. They only enforce the evaluation of the their arguments.

This lecture we will see how a program can be automatically

transformed into an equivalent one with iterative behavior. Next lecture we will see how a language can be defined/implemented in such a way that the definitional/implementation framework (Maude in our case) does not need to maintain any implicit control context.

Continuation-Passing Style Transformation

Compilers can generate very efficient code for programs admitting iterative control behavior, because function calls, recursive or not, can be reduced to just memory allocation for arguments followed by simple jumps in RAM.

In what follows we will define a procedure, called *CPS* as an abbreviation for *continuation-passing style*, that transforms any program in our functional language into a program which has iterative control behavior. The trick is to

Collect and pass, as an additional argument to every function, all the control information needed to continue the execution of the program after returning from a call to that function.

Therefore, instead of having a function evaluate and then *return* its value to the calling context, we will *pass the calling context* to the function, so the function will "know" what to do with its value after it evaluates. The additional argument is called *continuation*.

The CPS transformation can be seen as a pre-compilation program transformation process, which arranges the programs in a form admitting very efficient compilation. We do not discuss compilation issues here, but because of its practical and theoretical importance we will discuss and define the CPS transformation procedure.

Several functional languages are compiled after an initial CPS transformation. However, despite the fact that even very serious languages, such as New Jersey's Standard ML, use CPS before

compilation, there is *not* a full agreement on whether the CPS transformation really brings a huge benefit in compilation.

The reason is that the additional continuation arguments passed to each function can grow quite large and thus become a serious bottle-neck of the entire technique. To keep the continuations compact, many compilers use aggressive lambda-optimizations.

There is an interesting research initiative in using continuations in *internet applications*, such as client-server systems, based on the idea that a server maintains a continuation for each client, knowing how to continue the service.

Tail Positions

Recall that the second version of the factorial function has not required control context information to be stored because the result of the recursive call was the result of the entire computation. We call such function calls, whose result is the value of the entire expression, *tail calls*.

An expression is called *simple* if and only if it does not contain any function call except in the body of other function declarations. That guarantees that the evaluation of a simple expression will never encounter a function call; note that function declarations are evaluated to closures, so their bodies are not evaluated (unless the function is called, in which case the expression is not simple anyway). Primitive operations, such as addition, car, cdr, etc., are not considered function calls. A function declaration is always simple and a function invocation is never simple.

A subexpression is in *tail position* in an expression if and only if it has the property that if it is evaluated then its result becomes the result of the entire expression. An expression may have more than one tail position; e.g., the conditional. For our functional language (to keep the presentation simple, we only consider the default call-by-value and do not consider side effects), the tail positions are given by the following easy to remember rules, where `Tail` denotes a tail position:

```
if <Exp> then Tail else Tail
let / let rec <Bindings> in Tail
fun (<Parameter>) Tail
```

Function calls on tail positions are called *tail calls*. An expression is in *tail form* if and only if it has only simple expressions in non-tail positions. Our goal next is to transform any functional program into an equivalent one which is in tail form.

An Example

Let us consider the recursive behavior version of the factorial. The CPS procedure defined next will *automatically* transform it into an equivalent tail-form expression similar to the one below:

```
let rec f n k = if n eq 0
                then k 1
                else f (n - 1) (fun v -> k (v * n))
            in f 5 (fun x -> x)
```

The idea is to add one more argument to each function, called *the continuation argument*, with the continuation-based intuition that it collects all the control information needed in order for a function call to be moved from a non-tail position to a tail-position. This way, a function call will never need to *return*; a function, instead, will *pass* its result to the continuation taken as last argument.

The CPS Procedure

The CPS program transformation will be defined by means of two transformations defined mutually recursively:

- One, written `<E>`, for detecting all the function declarations in a simple expression `E` and transforming them by adding the continuation parameter;
- Another, written `<E => K>`, for translating an expression `E`, that normally evaluates to some value `v`, into one that passes `v` to the continuation `K`. Note that `K` is nothing but a function declaration.

`<_>` and `<_=>_>` will be defined in the sequel.

`<E>` traverses the expression `E`, which is expected to be simple, and transforms any function declaration `fun P -> E` into

```
fun P K -> <E => K>
```

where `K` is a fresh name, reflecting the intuition that each function will take a continuation argument to which it will

"pass" its result once calculated.

The transformation `<E => K>` is the difficult one to define. With the intuition that the expression must be modified to one which is in tail form and which passes its value to the continuation, we can split its definition into the following four possibilities.

(1) If `E` is a *simple expression* then

`<E => K>` rewrites to `K <E>`.

Therefore, the simple expression is first transformed and then "passed" to the continuation.

For the remaining 3 cases, let us assume that `E` is *not* simple.

(2) Suppose that all the direct subexpressions of `E` are simple. The only way for that to be the case is that `E` is a function call. Suppose therefore that `E` is `F E'`, where `F` and `E'` are both simple. Then

`<F E' => K>` rewrites to `<F> <E'> K`.

Therefore, all the direct simple subexpressions are first transformed and then the continuation is "passed" as an additional argument. Since all the function declarations are eventually transformed into ones admitting one more argument for the continuation, the transformation above will be globally well defined.

For the remaining two cases, let us assume that `E` has at least one direct subexpressions that is *not* simple. Such a subexpression can occur either on a non-tail position or on a tail position.

(3) If `E` has some direct subexpression on a *non-tail position* which is *not* simple, then the normal evaluation of `E` would generate a function call and therefore a control context issue. To eliminate this problem, one can instead transform the non-simple non-tail direct subexpression, say `E'`, to send its result to an appropriately modified continuation, and then evaluate `E'` instead of `E`. More precisely, suppose that `E` has the form `C[E']` (`C` is called a *context* in the literature). Then

`<C[E'] => K>` rewrites to `<E' => (fun v <C[v] => K>)>`,

where `v` is a fresh name. Note that the right-hand term contains two applications of `<_=>_>`. However, the inner one is applied to an expression which is "simpler" than the original one (because one of its non-simple direct subexpressions has been replaced by a name), while the outer one is applied on an expression, `E'`, that is smaller than the original one. These will ensure that the mutual recursion terminates.

There is one complication here: `let rec` declares new names that can be seen by its direct non-simple non-tail subexpressions. Therefore, one cannot safely apply the CPS transformation on `let rec` expressions. Consider, for example, that `E` is the expression

```
let rec x = f x
in x
```

as part of a larger context. Then, if one just applies blindly the transformation above, where `E'` is the non-simple non-tail `f x`, then one gets that `<E => K>` is

```
f x (fun v -> K (let rec x = v in x))
```

The problem here is that the argument `x` of `f` is now out of its scope, so the program may be either undefined or evaluate to a wrong value.

Side effects and loops present additional complications to the simple CPS transformation presented in CS422. More advanced

versions on CPS transformations will be covered in CS522 next semester. For the time being, we remove all the imperative features of `FUN` and the `let rec` will be only allowed to bind names to *simple* expressions (note that this will still allow us to define recursive functions, so the major role of `let rec` is not affected!).

The three rules above reduce any transformation to one in which the only non-simple expressions are on tail positions:

(4) If none of the above transformations can be applied anymore then the only thing left to do is to apply `<_ => K>` to the direct subexpressions of `E` occurring on tail positions and to also propagate `<_>` appropriately on the non-tail simple direct subexpressions. There are two concrete cases to analyze:

`<if B then E1 else E2 => K>` rewrites to
`if <B> then <E1 => K> else <E2 => K>`,

and

```
[let/let rec X1=E1 and ...  and Xn=En in E => K]
```

rewrites to

```
let/let rec X1=<E1> and ...  and Xn=<En>
in <E => K>,
```

Note that `B` in the former and `E1,...,En` in the later are all simple, otherwise rule *(3)* would apply instead.

There is a slight complication here because of the new bindings generated by `let` and `let rec`, that may capture free names occurring in `K` (coming from the outer context). We actually need to generate a fresh name, say `k`, and then to rewrite the above to:

```
let/let rec k=K and X1=<E1> and ...  and Xn=<En>
in <E => k>
```

The CPS transformation of an expression `E`, say `cps(E)`, can be now defined as `fun k -> <E => k>`, where `k` is some fresh name. With this, `E` is then equivalent to `cps(E) (fun x -> x)`.

An Example CPS Transformation

You may be asked at the final to apply the CPS transformation by hand to a simple program like the one below, which removes all the elements of a list which are equal to a certain integer number:

```
let rec r n l = if null?(l) then []
                else if n eq car(l)
                     then r n cdr(l)
                     else cons(car(l), (r n cdr(l)))
in r 3 [3 :: 1 :: 3 :: 2 :: 3 :: 3 :: 3 :: 4 :: 3 :: 5 :: 3 :: 3]
```

The result is (equivalent to) the following, which is also the result that your CPS definition should return (modulo currying, etc.):

```
let rec r n l k = if null?(l) then k []
                  else if n eq car(l)
                       then r n cdr(l) k
                       else r n cdr(l) (fun v -> k cons(car(l), v))
in r 3 [3 :: 1 :: 3 :: 2 :: 3 :: 3 :: 3 :: 4 :: 3 :: 5 :: 3 :: 3]
          (fun x -> x)
```

The Homework Exercise

There are three Maude files posted together with this lecture:

- `fun-definition-for-cps.maude` contains a complete simplified definition of `FUN`, with no imperative features and just call-by-value; you should not need to modify this file.
- `fun-cps.maude` contains an *incomplete* definition of the CPS transformation technique described in this lecture;
- `fun-cps-examples.maude` contains lots of examples on which you can test your definition.

Homework Exercise 1 *Complete the definition of the CPS transformation in* `fun-cps.maude`.

This is a rather longer and time consuming exercise. Make sure that you read the comments in the current `fun-cps.maude`

carefully. Since Maude is a specification language and therefore has no side effects, one needs to make sure that the state is always modified and propagated explicitly by your operators. In particular, `<E>` and `<E => K>` will become `< E,S >` and, respectively, `< E => K, S >` in Maude. Note the spaces separating the symbols `<` and `>` from the variables. Unlike brackets, we need to allow spaces around these symbols.